CS170 Solutions to the First Midterm

1. **(2 points each question) True or false?** Circle the correct answer. No explanation required. No points will be subtracted for incorrect answers, so guess all you want.

T The solution to the recurrence $T(n) = 5T(\frac{n}{3}) + n^3$ is $\Theta(n^3)$.
The master theorem yields this at once. Or you could expand out the recurrence.

T The solution to the recurrence $T(n) = 4T(\frac{n}{2}) + n^2 \log n$ is $\Theta(n^2 \log^2 n)$.
If you plug in this solution, you will find that it works. Alternatively, you could directly use our two other techniques.

F $\lceil \log n \rceil !$ is $O(n^2)$ (*Note:* ! denotes factorial)
$\lceil \log n \rceil !$ consists of $\log n$ terms, most of which are considerably larger than four. Therefore, it will be orders of magnitude bigger than $4^{\log n} = n^2$.

F $n^{\sin \frac{\pi}{2} n}$ is $O(\sqrt{n})$
As n varies, $\sin \frac{\pi}{2} n$ oscillates between -1 and 1. Thus the expression is infinitely often equal to n, which is not $O(\sqrt{n})$.

T The cubes of the 63rd roots of unity are the 21st roots of unity.
For the FFT, we needed the fact that the squares of the $2n$th roots of unity are the nth roots of unity. This follows by similar reasoning.

F If n is not prime, then for any positive integer a which is less than n and relatively prime to n, $a^{n-1} \neq 1 \pmod{n}$.
For instance, $1^{n-1} = 1 \pmod{n}$.

T Carmichael numbers are never prime.
Carmichael numbers constitute bad cases for the simple primality tester presented in class.

T $27^{45} = 82^{85} (\mathrm{mod} 55)$
$\phi(55) = 40$ and $27 = 82 \pmod{55}$, so $27^{45} = 27^5 \pmod{55}$ and $82^{85} = 27^{85} = 27^5 \pmod{55}$.

F There exist integers x and y such that $273x + 42y = 7$.
Three divides the expression on the left but not the number on the right.

T If $f = O(g)$ then $f^2 = O(g^2)$.
If $\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c$ for some constant c, then $\lim_{n\to\infty} \frac{f(n)^2}{g(n)^2} \leq c^2$.

F If $f = O(g)$ then $2^f = O(2^g)$.
For instance $2n = O(n)$ but $2^{2n} \neq O(2^n)$.

2. (**10 points**) Alice wants to write a message that looks like it was digitally signed by Bob. She notices that Bob's public RSA key is (17, 391). To what exponent should she raise her message? (Answer of the form $x^{-1} mody$ gets half credit.)

Solution: When Bob signs a message, he computes $M^d \bmod n$, where (d, n) is his private key. Recall that if $n = pq$, then $d = e^{-1} \bmod (p-1)(q-1)$, where (e, n) is Bob's public key.

Now, notice that $391 = 17 \cdot 23$, and so $d = 17^{-1} \bmod (17-1)(23-1) = 17^{-1} \bmod 352$. To compute this, we use the Extended Euclid algorithm, with inputs 17 and 352. Because 17 and 352 are relatively prime, Euclid's will yield a pair (x, y) such that $17x + 352y = 1$. In particular, we will have $x = 145$ and $y = -7$, hence $17 \cdot 145 \bmod 352 = 1$, and so $d = 17^{-1} \bmod 352 = 145$. *Sergey Ioffe graded this problem.*

3. (**10 points**) Give a brief description of an efficient (polynomial time) algorithm for computing $a^{(b^c)} (\mathrm{mod} p)$ for p prime. Briefly explain why it is correct and why it is polynomial.

   Solution: For your general edification, here is a much more detailed time analysis than you needed to give:

   If a is divisible by p, then $a^{b^c} \bmod p = 0^{b^c} \bmod p = 0$. In what follows we assume that p doesn't divide a.

   By the Fermat's Little Theorem, $a^{p-1} \bmod p = 1$. Therefore, $a^{b^c} \bmod p = a^{b^c \bmod (p-1)} \bmod p$.

   Let us first compute $b^c \bmod (p-1)$. This can be done efficiently by using repeated squaring. Recall from class that this will involve $O(\log c)$ multiplications of numbers between 0 and p, and taking the results modulo $p-1$. Each of these operations takes $O(\log^2 p)$ time. Thus, $b^c \bmod (p-1)$ can be computed in $O(\log c \cdot \log^2 p + \log b \cdot \log p)$ time (the extra $\log a \cdot \log p$ accounts for the initial computation of $b \bmod (p-1)$).

   Now, we need to compute $a^{b^c \bmod (p-1)} \bmod p$. This can be done as above, using repeated squaring. Notice that $b^c \bmod (p-1) < p$, thus the exponentiation will take $O(\log p \cdot \log^2 p + \log a \cdot \log p)$ time.

   The total running time of this algorithm is $O(\log c \cdot \log^2 p + \log b \cdot \log p + \log^3 p + \log a \cdot \log p) = O(n^3)$, where n is the number of bits in the input.

   Sergey Ioffe graded this problem.

4. **(10 points)** Alice wishes to send 20 integers (expressed modulo p for some large prime p) to Bob over a noisy channel. She knows that at most 12 of the integers she sends will be corrupted during transmission.

   - **(5 points)** Using the Berlekamp-Welch error correction scheme, Alice lets the 20 integers be the values of a polynomial at the points 1 through 20. What is the minimum number of integers she should send to insure that Bob can recover the original 20 integers?

     Solution: Since there are 20 integers, a polynomial of degree $d = 19$ should be used. The number of mistakes that we need to accommodate is $k = 12$. We saw in class that if we send n integers then

     $$k \leq \frac{n - (d+1)}{2}$$

     whereby we need $n \geq 44$.

   - **(5 points)** After Bob correctly recovers the polynomial, he loses two of the coefficients. How many polynomials, as a function of p, are consistent with the information that Bob now has?

     Solution: Each of the two missing coefficients can have any value in the range $0, \ldots, p-1$. Therefore there are p^2 choices.

   Sanjoy Dasgupta graded this problem.

5. (**15 points**) Recall Shamir's Secret Sharing scheme discussed in class.

- (**3 points**) Two street gangs, the Cool Coders and the Overloaded Operators, wish to equally share the password to their CS170 account. Outline a simple secret sharing scheme that insures that both parties must input their shares before they can log in.
  Solution: Let $p(x) = ax + b$ be a polynomial of degree one with coefficients in $GF(p)$, for p a large enough prime, say $p = 23$. Give $p(1)$ to the Cool Coders and $p(2)$ to the Overloaded Operators. Let the secret be $p(0)$. Since a degree one polynomial is determined by its values at two distinct points, but not by its value at one point, both parties have to input their shares before they can log in.
- (**12 points**) Now suppose that the members of the two groups do not trust their fellow members either. What they need is a scheme whereby any three of the seven Cool Coders together with any four of the nine Overloaded Operators would be able to log in, yet any pair of sets that has fewer members from either side has no information about the key —no matter how many members of the other side are present. Devise a secret sharing scheme that works for these specifications.

  Solution: Again, we will use polynomials with coefficients in $GF(23)$. Let $q(x)$ be a polynomial of degree two and give the ith member of the Cool Coders $q(i), 1 \leq i \leq 7$. Note that since $p = 23$, this means that we evaluate $q(x)$ at seven distinct points. Then since a degree two polynomial is uniquely determined by its values at three distinct points, any three of the Cool Coders can calculate $q(0)$, but not less than three. Let $r(x)$ be a polynomial of degree three and give the jth member of the Overloaded Operators $r(j), 1 \leq j \leq 9$. Again, since p is 23, every Overloaded Operator gets $r(x)$ evaluated at a different point. Since a degree three polynomial is uniquely determined by its values at four distinct points, any four of the Overloaded Operators can calculate $r(0)$, but not less than four. Then by the previous part both $q(0)$ and $r(0)$ are needed to calculate the degree one polynomial $p(x)$ that goes through $(1, q(0))$ and $(2, r(0))$. So if we let the secret be $p(0)$, we need three Cool Coders and four Overloaded Operators to log in.

  Kirsten Eisentraeger graded this problem.

6. **(15 points)**An array is said to have a *majority element* if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element.

   - **(2 points)** *Briefly describe how this can be accomplished in linear time using median-finding.*

   Solution: If there is a majority element in the array it will be the median. Thus, just run the linear time median finding algorithm, and compare the result with all the elements of the array (also linear time). If $\frac{n}{2}$ elements are the same as the median, the median is a majority element. If not, there is none.

   Suppose now that the elements of the array are not from some ordered domain like the integers, and so there can be no comparisons of the form "is the ith element of the array greater than the jth element of the array?" However you can answer questions of the form: "Are the ith and jth elements of the array the same?" in constant time. Such queries constitute the only way whereby you can access the array. (Think of the elements of the array as GIF files, say.) Notice that your solution above cannot be used now.

(13 points) Give a $O(n \log n)$ time divide-and-conquer algorithm for this task.

A. Brief description of the algorithm.

Solution: The following algorithm outputs the majority element if there is one, and outputs FALSE otherwise:

Given an array N of size n, divide it into two arrays, A and B, of size $\frac{n}{2}$.

Now given solutions to the two sub-problems, we merge them as follows:

If both A and B have majority elements and they're equal, then we output that element as the majority element for N. If neither has a majority element, we output FALSE. If only one of the sub-arrays has a majority element, or if they have different majority elements, we compare it/them to each of the elements of N in turn, keeping count of the number of matches. If one of these "sub-majority" elements occurs more than $\frac{n}{2}$ times, we output it as the majority element for N. If not, then we output FALSE.

B. Justification of correctness.

If m is a majority element for N, it must necessarily be a majority element for at least one of A,B.

C. Running time and justification.

Merging subproblems involves at most $2n + 1$ comparisons. Thus, our recurrence is

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

which yields a running time of $O(n \log n)$ by the Master theorem.

Sara Robinson graded this problem.

Grading criteria: If you showed you understood divide-and-conquer but didn't give a correct merge you got 2-4 points. If you computed a correct running time for your incorrect algorithm you got a few additional points.

Many of you gave a randomized algorithm, similar to median finding, which has n^2 worst-case time but linear expected time. I gave a lot of partial credit for this answer (depending on the clarity of the explanation) although it's not a divide-and-conquer algorithm.

An algorithm no better than doing all possible comparisons in n^2 time got 0 points.

Extra Credit: Give a linear time algorithm for this task.

(*Note:* No partial credit. Attempt this problem *only* if you have time to spare.)

Solution: Create a stack and push a_1, the first value of the array, onto it. Now for each position $1 < i \leq n$ of the array, compare a_i with the top element of the stack, t. If $a_i = t$, then push a_i onto the stack and increment i. If $a_i \neq t$, then increment i and pop t off the stack.

At the end, If the stack is empty, output FALSE. If not, then output the top element of the stack.

Sara Robinson graded this problem.